IEEE 754 Standard for representing Floating Point Numbers

Abstract

This is a documentation about the basic operations involving floating point numbers, and its computer implementation. It would specifically deal with the IEEE 754 Standard for representing floating point numbers. The simulation of the different operations performed for representation and modification of floating point numbers was done in C++, and the code snippets from the same are shown in the document.

The problem in representing real numbers

It's really easy to write integers as binary numbers in two's complement form. It's a lot more difficult to express floating point numbers in a form that a computer can understand. The biggest problem, of course, is keeping track of the decimal point. There are lots of possible ways to write floating point numbers as strings of binary digits, and there are many things to consider when picking a standard method to do this:

- **Range**: To be useful, your method should allow very large positive and negative numbers.
- **Precision**: Can you tell the difference between 1.7 and 1.8? How about between 1.700001 and 1.700002? How many decimal places should you remember?
- **Time Efficiency**: Does your solution make comparisons and arithmetic operations fast and easy?
- **Space Considerations**: An extremely precise representation of the square root of 3 is generally a wonderful thing, unless you require a megabyte to store it.
- **One-to-one Relationships**: Your solution will be a lot simpler if each floating-point number can be written only one way, and vice versa.

The IEEE 754 Form of representing a floating point number

The method that the developers of IEEE 754 Form finally hit upon uses the idea of scientific notation. Scientific notation is a standard way to express numbers; it makes them easy to read and compare. You're probably familiar with scientific notation with base-10 numbers. You just factor your number into two parts: a value whose magnitude is in the range of $1 \le n < 10$, and a power of 10. For example:

$$3498523 \text{ is written as } 3.498523 \times 10^{6}$$

$$-0.0432 \text{ is written as } -4.32 \times 10^{-2}$$

The same idea applies here, except that you need to use powers of 2 because the computer works efficiently with binary numbers. Just factor your number into a value whose magnitude is in the range $1 \le n < 2$, and a power of 2.

$$-6.84 \text{ is written as } -1.71\times22$$
$$0.05 \text{ is written as } 1.6\times2^{-5}$$

To create the bit string, we need to massage this product so that it takes the following form:

$$(-1)^{\text{sign bit}} \times(1+\textbf{fraction}) \times 2^{\text{exponent - bias}}$$

Once this is done, we will have three key pieces of information (shown in color above) that, when taken together, identify the number:

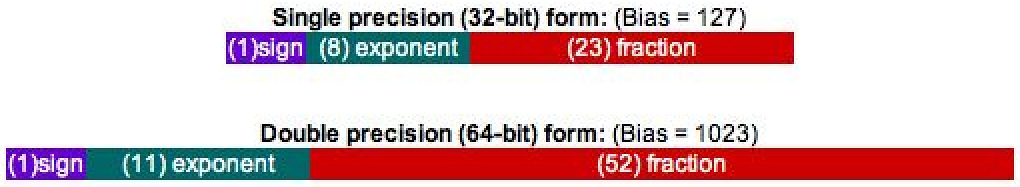
- *First Piece* -- If the sign bit is a 0, then the number is positive; $(-1)^0=1$. If the sign bit is a 1, the number is negative; $(-1)^1=-1$.
- *Second Piece* -- We always factor so that the number in parentheses equals (1+ some fraction). Since we know that the 11 is there, the only important thing is the fraction, which we will write as a binary string.

- If we need to convert from the binary value back to a base-10 value, we just multiply each digit by its place value, as in these examples:

$$(0.1)_{\text{binary}}=2^{-1}=0.5$$
$$(0.01)_{\text{binary}}=2^{-2}=0.25$$
$$(0.101)_{\text{binary}}=2^{-1}+2^{-3}=0.625$$

- *Third Piece* -- The power of 2 that you got in the last step is simply an integer. Note, this integer may be positive or negative, depending on whether the original value was large or small, respectively. We'll need to store this exponent -- however, using the two's complement, the usual representation for signed values, makes comparisons of these values more difficult. As such, we add a constant value, called a *bias*, to the exponent. By biasing the exponent before it is stored, we put it within an unsigned range more suitable for comparison.
    - For single-precision floating-point, exponents in the range of -126 to + 127 are biased by adding 127 to get a value in the range 1 to 254 (0 and 255 have special meanings).
    - For double-precision, exponents in the range -1022 to +1023 are biased by adding 1023 to get a value in the range 1 to 2046 (0 and 2047 have special meanings).
- The sum of the bias and the power of 2 is the exponent that actually goes into the IEEE 754 string. Remember, the exponent = power + bias. (Alternatively, the power = exponent-bias). This exponent must itself ultimately be expressed in binary form -- but given that we have a positive integer after adding the bias, this can now be done in the normal way.

When you have calculated these binary values, you can put them into a 32- or 64-bit field. The digits are arranged like this:


Single precision (32-bit) form: (Bias = 127)
(1)sign
(8) exponent
(23) fraction
Double precision (64-bit) form: (Bias = 1023)
(1)sign
(11) exponent
(52) fraction

By arranging the fields in this way, so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits -- the resulting value will actually be ordered properly for comparisons, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.

There are some special cases:

- *Zero*
- Sign bit = 0; biased exponent = all 0 bits; and the fraction = all 0 bits;
- *Positive and Negative Infinity*
- Sign bit = 0 for positive infinity, 1 for negative infinity; biased exponent = all 1 bits; and the fraction = all 0 bits;
- *NaN (Not-A-Number)*
- Sign bit = 0 or 1; biased exponent = all 1 bits; and the fraction is anything but all 0 bits. (NaN's pop up when one does an invalid operation on a floating point value, such as dividing by zero, or taking the square root of a negative number.)

Converting to IEEE 754 Form

Suppose we wish to put 0.085 in single-precision format. Here's what has to happen:

1. *The first step is to look at the sign of the number.*

   Because 0.085 is positive, the sign bit = 0.
2. *Next, we write 0.085 in base-2 scientific notation*

   This means that we must factor it into a number in the range $(1 \le n < 2)$(1≤n<2) and a power of 2.

$$0.085 == (-1)^0 (1+\text{fraction}) \times 2^{\text{power}}, \text{ or, equivalently:}$$

$$0.085/2^{\text{power}} = 1+\text{fraction}$$

   As such, we divide 0.085 by a power of 2 to get the $(1+\text{fraction})$(1+fraction):

0.085/2−1=0.17

0.085/2−2=0.34

0.085/2−3=0.68

0.085/2−4=1.36

Therefore, 0.085=1.36×2−40.085=1.36×2−4

3. *Now, we find the exponent*

   The power of 2 used above was -4, and the bias for the single-precision format is 127. Thus,

   $$\text{exponent}=-4+127=123=(01111011)_{\text{binary}}$$

4. *Then, we write the fraction in binary form*

   Successive multiplications by 2 (while temporarily ignoring the unit's digit) quickly yields the binary form:

```
0.36 x 2 = 0.72
0.72 x 2 = 1.44
0.44 x 2 = 0.88
0.88 x 2 = 1.76
0.76 x 2 = 1.52
0.52 x 2 = 1.04
0.04 x 2 = 0.08          Once this process terminates or starts repeating,
0.08 x 2 = 0.16          repeating, we read the unit's digits from top to
0.16 x 2 = 0.32          bottom to reveal the binary form for 0.36:
0.32 x 2 = 0.64
0.64 x 2 = 1.28             0.0101110000101000111101011 1000...
0.28 x 2 = 0.56
0.56 x 2 = 1.12
0.12 x 2 = 0.24
0.24 x 2 = 0.48
0.48 x 2 = 0.96
0.96 x 2 = 1.92
0.92 x 2 = 1.84
0.84 x 2 = 1.68
0.68 x 2 = 1.36
0.36 x 2 =  ...  (at this point the list starts repeating)
```

   As you can see, 0.36 has a a non-terminating, repeating binary form. This is very similar to how a fraction, like 5/27 has a non-terminating, repeating decimal form. (i.e., 0.185185185...)

   However, single-precision format only affords us 23 bits to work with to represent the fraction part of our number. We will have to settle for an approximation, rounding things to the 23rd digit. One should be careful here -- while it doesn't happen in this example, rounding can affect more than just the last digit. This shouldn't be surprising -- consider what happens when one rounds in base 10 the value 123999.5 to the nearest integer and gets 124000. Rounding the infinite string of digits found above to just 23 digits results in the bits 0.01011100001010001111011.

   (*Note, we round "up" as the binary value 0.0111000... is greater than the decimal value 0.05.*)

   This rounding that we have to perform to get our value to fit into the number of bits afforded to us is why floating-point numbers frequently have some small degree of error when you put them in IEEE 754 format. It is very important to remember the presence of this error when using the standard Java types (float and double) for representing floating-point numbers!

5. *Finally, we put the binary strings in the correct order.*

   Recall, we use 1 bit for the sign, followed by 8 bits for the exponent, and 23 bits for the fraction.

   So 0.85 in IEEE 754 format is:

   0 01111011 01011100001010001111011

This process of converting a floating point number to a bit array was implemented in C++ by using a variable of a structured data type containing 3 data members, namely sign, exponent, and mantissa. The sign part was an integer, whereas the exponent and mantissa parts were integer arrays (even datatypes such as short or boolean can be used, as long as it can store 0's and 1's). The floating point can be entered from the console, and this is processed to a 3 sets of bit arrays, which are then stored in a structured variable. The code is as follows :-

// **To define a structured data type consisting of 3 components - sign, exponent, and mantissa**

```
struct floatComp{
  int sign;
  int exp[8];
  int man[23];
};
```

// **To represent a real number in the form of a structured variable of type floatComp**

```
void floatRep(floatComp &no,float x){
  unsigned int a=*((unsigned int *)&x);
  for(int i=31;i>=0;i--)
  {
    int j=pow(2,i);
    cout<<(int)(a/j);
    if(i==31 || i==23)
      cout<<' ';
    if(i==31)                          //MSB holds sign of number
      no.sign=a/j;
    else if(i>22)                      //Next 8 bits represents the exponent value
```

```
        no.exp[30-i]=a/j;

      else                                     //Rest of the bits store the value of the mantissa
        no.man[22-i]=a/j;
      a%=(unsigned int)pow(2,i);    //To process each of the 32 bits used for real number
   }
}
```

Converting from IEEE 754 Form

Suppose we wish to convert the following single-precision IEEE 754 number into a floating-point decimal value:

11000000110110011001100110011010

1. *First, we divide the bits into three groups:*

   1   10000001   10110011001100110011010

   The first bit shows us the sign of the the number.
   The next 8 bits give us the exponent.
   The last 23 bits give us the fraction.

2. *Now we look at the sign bit*

   If this bit is a 1, the number is negative; if it is 0, the number is positive. Here, the bit is a 1, so the number is negative.

3. *Next, we get the exponent and the correct bias*

   To get the exponent, we simply convert the binary number 10000001 back to base-10 form, yielding 129

   Remember that we will have to subtract an appropriate bias from this exponent to find the power of 2 we need. Since this is a single-precision number, the bias is 127.

4. *Then we must convert the fraction bits back into base 10*

   To do this, we multiply each digit by the corresponding power of 2 and sum the results:

$$\begin{aligned} 0.10110011001100110011010_{binary} &= 1\cdot 2-1+0\cdot 2-2+1\cdot 2-3+1\cdot 2-4+0\cdot 2-5+\cdots \\ &= 1/2+1/8+1/16+\cdots \\ &= 0.7000000476837158 \end{aligned}$$

Remember, this number is most likely just an approximation of some other number. There will most likely be some error.

5. *We have all the information we need. Now we just calculate the following expression:*

$$(-1)^{\text{sign bit}}(1+\text{fraction})\times 2^{\text{ exponent - bias}}=(-1)^{1}(1.7000000476837158)\times 2^{129-127}$$
$$=-6.800000190734863$$

Thus, the IEEE 754 number 11000000110110011001100110011010 gives the floating-point decimal value -6.800000190734863. It is reasonable to suspect that the original number stored was probably -6.8, although this would be hard to prove... (One can verify that -6.8 does result in the exact same bit string, however.)

This process of converting a bit array to a floating point number was implemented in C++ by using a variable of the structured data type floatComp. The 3 sets of bit arrays are processed, and the resulting values are combined and displayed as a floating point value. The code is as follows :-

// To extract and return the real number represented by the structured variable of type floatComp

```
float floatNum(floatComp no)
{
  unsigned int x=0;

  //To display the computer representation of the real number
  cout<<"\nRepresentation of resulting float - ";
  for(int i=0;i<32;i++)
  {
    // To add all the bits to an unsigned int variable one by one
    x*=2;
    if(i==0){
      x+=no.sign;
      cout<<no.sign<<' ';
    }
    else if(i<9){
      x+=no.exp[i-1];
      cout<<no.exp[i-1];
    }
```

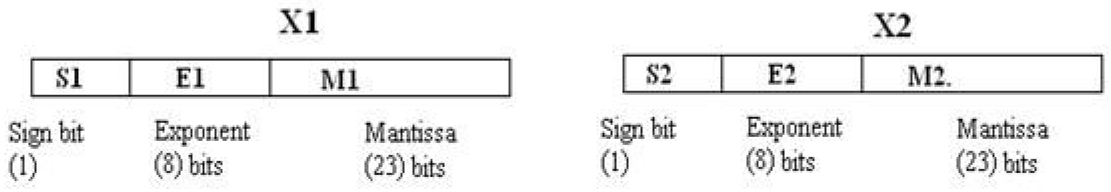
```
      else{
        x+=no.man[i-9];
        cout<<no.man[i-9];
      }
      if(i==8)
        cout<<' ';

  }
  cout<<endl;
```

//Return the floating point equivalent of the computer representation of the unsigned int data

```
  return(*(float *)&x);
}
```

IEEE 754 standard floating point Arithmetic

Let us look at **Multiplication**, **Addition**, **subtraction** & **inversion** algorithms performed on IEEE 754 floating point standard. Let us consider the IEEE 754 floating point format numbers X1 & X2 for our calculations.



IEEE 754 standard floating point Addition Algorithm

Floating-point addition is more complex than multiplication, brief overview of floating point addition algorithm have been explained below :

$$X3 = X1 + X2$$
$$X3 = (M1 \times 2^{E1}) +/- (M2 \times 2^{E2})$$

1) X1 and X2 can only be added if the exponents are the same i.e

$$E1=E2.$$

2) We assume that X1 has the larger absolute value of the 2 numbers. Absolute value of of X1 should be greater than absolute value of X2, else swap the values such that Abs(X1) is greater than Abs(X2).

   Abs(X1) > Abs(X2).

3) Initial value of the exponent should be the larger of the 2 numbers, since we know exponent of X1 will be bigger , hence

   Initial exponent result E3 = E1.

4) Calculate the exponent's difference i.e.

   Exp_diff = (E1-E2).

5) Left shift the decimal point of mantissa (M2) by the exponent difference. Now the exponents of both X1 and X2 are same.

6) Compute the sum/difference of the mantissas depending on the sign bit S1 and S2.

   If signs of X1 and X2 are equal (S1 == S2) then add the mantissas
   If signs of X1 and X2 are not equal (S1 != S2) then subtract the mantissas

7) Normalize the resultant mantissa (M3) if needed. (1.m3 format) and the initial exponent result E3=E1 needs to be adjusted according to the normalization of mantissa.

8) If any of the operands is infinity or if (E3>Emax) , overflow has occurred ,the output should be set to infinity. If(E3 < Emin) then it's a underflow and the output should be set to zero.

9) Nan's are not supported.

NOTE: For floating point Subtraction, invert the sign bit of the number to be subtracted and apply it to floating point adder

To implement this in code, each field of the floating point number has to be dealt with separately, as separate numbers. Given the previous code shown, it would be hard to work on variables of the data type floatComp directly, as the exponent and mantissa fields as arrays. So in order to avoid this, these fields are converted into integers, which can then be processed easily. Once processed, the resulting set of integers can then be converted back to arrays and stored in a floatComp variable, which is then displayed as the result. The code to convert the exponent and mantissa arrays of the floatComp variable into integers, and vice versa, are shown below :

// For finding the mantissa from a floatComp variable and multiplying it by a constant (2^23) so as to store it as an integer, adding the hidden bit (1) in the beginning

```
long long unsigned int findMan(floatComp &one)
{
   long long unsigned int x=1;
   for(int i=0;i<M;i++)
      x=(x*2)+one.man[i];
   return x;
}
```

// For storing the mantissa represented as an integer into a floatComp variable excluding the hidden bit (1)

```
void displayMan(floatComp &result,long long unsigned int res)
{
   cout<<"Resulting mantissa - ";
   for(int i=23;i>=0;i--)
   {
         long long unsigned int j=pow(2,i);
         if(i<23)
            result.man[22-i]=(int)(res/j);
         cout<<(int)(res/j);
         if(i==23)
            cout<<'.';
         res%=j;

   }cout<<endl;

}
```

// For finding the (biased) exponent from a floatComp variable by storing it as an integer

```
int findExp(floatComp &one)
{
   int x=0;
   for(int i=0;i<8;i++)
      x=(x*2)+one.exp[i];
```

```
    return x;
}
```

// For storing the biased exponent represented as an integer into a floatComp variable

```
void displayExp(floatComp &result,int x)
{
    cout<<"Resulting exponent - ";
        for(int i=7;i>=0;i--)
        {
            int j=pow(2,i);
            result.exp[7-i]=(int)(x/j);
            x%=j;
            cout<<result.exp[7-i];
        }
        cout<<endl;
}
```

The code for simulating the whole process of addition and subtraction of floating point numbers in the form of floatComp variables is as follows :-

// To add or subtract 2 floating point numbers, represented as floatComp variables

```
floatComp addab(int adsub,floatComp one,floatComp two){

    int x=findExp(one),y=findExp(two);
    int expdiff=x-y;                    //Checking which number has a greater exponent

    int m=findMan(one),n=findMan(two);

    //Making the exponents equal, by modifying the mantissa of the smaller number accordingly
    if(expdiff<=0){
        m/=(int)pow(2,y-x);
        x=y;
    }
```

```
else{
   n/=(int)pow(2,expdiff);
   y=x;
}

m*=pow(-1,one.sign);          //adsub represents addition or subtraction
n*=pow(-1,two.sign^adsub);         //  (0 -> +)  ;   (1 -> -)

int res;
res=m+n;                          //Adding the 2 integers (m,n)

if(res<0)                         //To check if the resulting sum is negative or not
{
   res*=-1;
   result.sign=1;
}
else result.sign=0;

cout<<"\nSign of result - "<<result.sign<<endl;

//For normalizing the exponent (May occur only if the result is not 0)
if(res!=0){

//If the 2 numbers have equal sign, then the sum may have a greater exponent. So the resulting
mantissa is reduced to have a fixed size, while the exponent is normalized accordingly
   if(one.sign==two.sign^adsub)
      while(res/pow(2,24)>=1 && x!=0)
      {
         res/=2;
         x++;
      }
//If the 2 numbers have opposite sign, then the sum may have smaller exponent. So the
resulting mantissa is increased to have a fixed size, while the exponent is normalized accordingly
   else
      while(res/pow(2,23)<1)
      {
            res*=2;
            x--;
        }
    }
```

//If the resulting mantissa is 0, then the final resulting floating point number will also be 0.

```
    else     x=0;
```

//In case the exponent goes out of bounds

```
    if(x>255)
        x=255;

    displayMan(result,res);
    displayExp(result,x);

    printf("\nResulting sum - %f",floatNum(result));
    return result;
}
```

IEEE 754 standard Floating point multiplication Algorithm

A brief overview of floating point multiplication algorithm have been explained below, X1 and X2.

$$\text{Result } X3 = X1 * X2$$
$$= (-1)^{S1} (M1 \times 2^{E1}) * (-1)^{S2} (M2 \times 2^{E2})$$

S1, S2 => Sign bits of number X1 & X2.
E1, E2: =>Exponent bits of number X1 & X2.
M1, M2 =>Mantissa bits of Number X1 & X2.

1) Check if one/both operands = 0 or infinity. Set the result to 0 or inf. i.e. exponents = all "0" or all "1".
2) The XOR operation is performed between S1 and S2, the signed bit of the multiplicand and the multiplier, respectively. The result is put into the resultant sign bit.
3) The mantissa of the Multiplier (M1) and multiplicand (M2) are multiplied and the result is placed in the resultant field of the mantissa (truncate/round the result for 24 bits).

$$M3=M1 * M2$$

4) The exponents of the Multiplier (E1) and the multiplicand (E2) bits are added and the base value is subtracted from the added result. The subtracted result is put in the exponential field of the result block.

$$E3=E1+E2-bias$$

5) Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
6) Check for underflow/overflow. If Overflow set the output to infinity & for underflow set to zero.

If $(E1 + E2 - bias) >= Emax$ then set the product to infinity.
If $(E1 + E2 - bias) <= Emin$ then set product to zero.

Performing division is similar to multiplication where the only difference is that the mantissa of the 2 numbers are divided, and the difference is found between the 2 exponents.

$$\text{Result } X3= X1 / X2$$
$$= (-1)^{S1} (M1 \times 2^{E1}) / (-1)^{S2} (M2 \times 2^{E2})$$

S1, S2 => Sign bits of number X1 & X2.
E1, E2: =>Exponent bits of number X1 & X2.
M1, M2 =>Mantissa bits of Number X1 & X2.

Unfortunately, the hardware to perform true division is pretty slow as division performed bitwise is far more complex than multiplication (it takes about double the time).

In order to improve upon the speed of the process, another method is adopted, called division by reciprocal approximation, which takes about the same time as a fl. pt. multiply. Here, instead of performing X3= X1 / X2, the multiplicative inverse of X2 is found (take it to be X2'), and the following is performed :

$$X3= X1 * X2'$$
$$= X1 * (1/X2)$$

This saves both time and space, as the same multiplication process can be performed for both multiplication and division.

Unfortunately, the results shown for division by reciprocal approximation are not always completely accurate as with true division as it is not always possible to get a perfectly accurate reciprocal. For example, taking the expression to be (3 / 3) :

True division: $3/3 = 1$ (exactly)

Reciprocal approx: $1/3 = .33333333$

$3 \times .33333333 = .99999999$, not 1

The code for simulating the whole process of multiplication and division of floating point numbers in the form of floatComp variables is as follows :-

```
// To multiply or divide 2 floating point numbers, represented as floatComp variables
floatComp multiplyab(int muldiv,floatComp one,floatComp two){

  //The resulting sign of the product will be XOR of the signs of the 2 numbers
  result.sign=one.sign^two.sign;
  cout<<"\nResult sign - "<<result.sign<<endl;

  if(muldiv)                          //muldiv represents either multiplication or division
    two=inverse(two);                         //   (0 -> *)  ;  (1 -> /)

  int normalize=0;

  long long unsigned int x=findMan(one),y=findMan(two),z;
  int a=findExp(one),b=findExp(two),res;

  //If any of the real numbers is 0, then the product is 0
  if(a==0 && x==0x00800000)
    x=0;
  if(b==0 && y==0x00800000)
    y=0;

  //To display the 2 mantissa
  long long unsigned int m=x,n=y;
  cout<<"\nMantissa of 1st number - ";
  for(int i=M;i>=0;i--)
  {
    int j=pow(2,i);
    cout<<m/j;
    if(i==M)
      cout<<'.';
    m%=j;
  }
```

```
    cout<<endl<<"Mantissa of 2nd number - ";

    for(int i=M;i>=0;i--)
    {
        int j=pow(2,i);
        cout<<n/j;

        if(i==M)
            cout<<'.';
        n%=j;
    }
    cout<<endl;

//Normal multiplication done on the integer representations of the 2 mantissa
    z=x*y;

    if((int)(z/pow(2,2*M+1))==1)        // In case mantissa is >= 2, normalization is done
    {
        z/=2;
        normalize++;
    }
    z/=pow(2,M);        //To reduce mantissa by M bits, since it originally had 2M bits

    //To calculate the resulting exponent as the sum of the 2 exponent values
    res=a+b-bias+normalize;        //To get a biased exponent

    //In case the exponent goes out of bounds
    if(res>255)
        res=255;
    if(res<0)
        res=0;
    displayMan(result,z);
    displayExp(result,res);
    cout<<"\nResulting product - "<<floatNum(result);
    return result;
}
```

```
// To find multiplicative inverse of a given number, represented by a floatComp
variable
floatComp inverse(floatComp two)
{
   int x=findExp(two);
   long long unsigned int m=findMan(two),extra=pow(2,23);

   //If the number is 0, it's inverse will be (+/-)infinity
   if(m==pow(2,23) && x==0)
   {
      x=255;
   }
   else{
      x=(2*bias)-x;       //To get the biased exponent of the inverse

      //If mantissa isn't 1, then the resulting exponent e' = (-e) - 1
      if(m>pow(2,23)){
         x-=1;
         extra*=2;
      }
      //Mantissa is given as -> m' = extra/m
      m=(extra*pow(2,23))/m;
   }
   cout<<endl;
   displayMan(two,m);
   displayExp(two,x);
   return two;
}
```

Transcendental Functions

A transcendental function is an analytic function that does not satisfy a polynomial equation, in contrast to an algebraic function. In other words, a transcendental function "transcends" algebra in that it cannot be expressed in terms of a finite sequence of the algebraic operations of addition, multiplication, and root extraction.

A few such functions which are going to be discussed in this document are the exponential function, the logarithm, trigonometric and hyperbolic functions.

Maclaurin Series

The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series

$$f(a)+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\cdots.$$

which can be written in the more compact sigma notation as

$$\sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n$$

where $n!$ denotes the factorial of n and $f^{(n)}(a)$ denotes the nth derivative of f evaluated at the point a. The derivative of order zero of f is defined to be f itself and $(x - a)^0$ and 0! are both defined to be 1. When $a = 0$, the series is also called a Maclaurin series.

The Maclaurin Series is very important in calculating transcendental functions as provides an approximate result for any differential function *f(x)* given a definite range, by having a series defined as shown above, and evaluating the same using basic arithmetic operations.

The Exponential Function

The exponential function is given by :

$$f(x) = e^x$$

where ‘e’ is an irrational real number constant, given as e=2.71828...

The exponential function e : C -> C can be characterized in a variety of equivalent ways. Most commonly, it is defined by the Maclaurin series, which finally evaluates to the following polynomial :

$$e^x = 1 + x + (x^2 /2!) + (x^3 /3!) + (x^4 /4!) + \ldots$$

Since the radius of convergence of this power series is infinite, this definition is applicable to all complex numbers x.

Upon further observations, it’s seen that e^x becomes more accurate for values of x close to 0, for a given number of terms in the power series.

The implementation of the above involves generating a finite number of terms (say around 20 terms) of the power series in order of increasing power, and adding them together to give the fial result. The code to calculate e^x is given below :

```
// To compare a floatComp structured variable with a real constant
int compare(floatComp, float);
// To add or subtract a floatComp variable with a floating point constant
floatComp intAdd(int, floatComp, float);
// To multiply or divide a floatComp variable with a floating point constant
floatComp intMultiply(int, floatComp, float);

// To find e^x, given a real number x
floatComp exponential(floatComp x)
{
   floatComp e,t;
   cout<<endl<<endl;
   floatRep(t,1);
   cout<<endl<<endl;
   floatRep(e,1);
   cout<<endl<<endl;

   //For gaining accurate results, the number is brought in the range of -2 <= x < 2
   //The value of e^x is updated accordingly

while(compare(x,-2)==-1)
   {
      e=intMultiply(1,e,54.5981500331);         //54.5981500331 = e^4
      x=intAdd(0,x,4);
   }
   while(compare(x,2)!=-1)
   {
      e=intMultiply(0,e,54.5981500331);
      x=intAdd(1,x,4);
   }

//For generating each term in the power series
   for(int i=1;i<21;i++)
   {
      t=multiplyab(0,t,x);     //t is multiplied by x each time, to have x^n in the nth term
      cout<<endl<<endl;
      cout<<endl<<endl;

      t=intMultiply(1,t,i);       //t is divided by i each time, to have 1/n! in the nth term
      cout<<endl<<endl;
      cout<<endl<<endl;

      e=addab(0,e,t);
   }
   cout<<"\n\n\te^x = "<<floatNum(e)<<endl;
   return e;
}
```

The Logarithm Function

The logarithm function is given by :

$$e^y = x$$

$$\text{So, } y = f(x) = \ln x = \log_e x$$

where ‘e’ is an irrational real number constant, given as e=2.71828...

The log function ln x : (0, inf) -> R can be characterized in a variety of equivalent ways. Most commonly, it is defined by the Maclaurin series, and for 0 < x <= 2, it finally evaluates to the following polynomial :

$$\ln x = t - (t^2 /2) + (t^3 /3) - (t^4 /4) + \ldots$$

$$\text{where } t = x-1$$

However, this equation holds true only for 0 < x <= 2. Therefore, we represent x as -

```
x = (2*2*2*2*...) * x'          ,where 0 < x <= 2
        So log x = log((2*2*2*2*...)*(x')))
                      = ((log 2)+(log 2)+(log 2)+(log 2)+...) + (log x')
```

Therefore, to calculate ln x, the value of x has to be brought between 0 and 2. This can be done by dividing the number continuously by 2, and increasing the value of the result by ln 2 accordingly.

The implementation of the above involves generating a finite number of terms (say around 20 terms) of the power series in order of increasing power, and adding them together to give the final result. The code to calculate ln x is given below :

```
// To find log x, given any non-negative real number x
floatComp logarithm(floatComp x){

   floatComp temp,t,ftemp;
   cout<<endl<<endl;
   floatRep(t,1);
   cout<<endl<<endl;
   floatRep(temp,0);
   cout<<endl<<endl;

   //If x=0, return -infinity
   if(compare(x,0)==0)
   {
      temp=intMultiply(0,temp,-1);
      temp=inverse(temp);

      return temp;
   }

   //To minimize errors in the series, an x' is chosen such that (1/1.9) <= x' <= 1.9
   while(compare(x,1.9)==1){
      temp=intAdd(0,temp,0.64185388617);
      x=intMultiply(1,x,1.9);
   }
   while(compare(x,0.52631578947)==-1){
      temp=intAdd(0,temp,-0.64185388617);
      x=intMultiply(0,x,1.9);
   }
   // Since the series is for log (x+1), x should be reduced by 1
   x=intAdd(1,x,1);

//For generating each term in the power series
   for(int i=1,si=1;i<21;i++,si*=-1)
   {
      t=multiplyab(0,t,x);
      cout<<endl<<endl;
      cout<<endl<<endl;

      temp=addab(0,temp,intMultiply(1,t,i*si));
      cout<<"\n\n-----------------------------------------------------------------\n\n\n";

   }
   return temp;
}
```

Trigonometric Functions

These set of functions include sin x, cos x, tan x, cosec x, sec x, cot x.

The sine of an angle A is defined as the ratio of the length of the side opposite to angle A to the length of the hypotenuse in a right angled triangle.

$$\sin A = \frac{\text{opposite}}{\text{hypotenuse}}$$

The cosine of angle A is defined as the ratio of the length of the side adjacent to angle A to the length of the hypotenuse in a right angled triangle.

$$\cos A = \frac{\text{adjacent}}{\text{hypotenuse}}$$

The tangent of angle A is defined as the ratio between the sine and cosine of A.

tan A = sin A / cos A

The other 3 functions, namely the cosecant, secant and cotangent of an angle, is the reciprocal of the sine, cosine, and tangent of the angle, respectively.

cosec A = 1 / sin A
sec A = 1 / cos A
cot A = 1 / tan A

Also, given angle A is in radians,

sin A = cos (pi - A) , where pi = 3.141592...

Thus, by finding sin A, the other trigonometric functions can be calculated easily.

The sine function sin x : R -> [-1, 1] can be characterized in a variety of equivalent ways. Most commonly, it is defined by the Maclaurin series, and it finally evaluates to the following polynomial :

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \qquad = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \qquad \text{for all } x$$

However, in order to improve upon time efficiency, sin x can be found as follows :

1) For -pi<=x<-1, sin x can be approximated as follows,

   sin x = -(0.5048 t^5)/5! + (0.8632*(t^4))/4! + (0.5048*(t^3))/3! - 0.8632*(t^2))/2! - - (0.5048*(t^1))/1! + 0.8632

2) For 1<x<=pi, sin x can be approximated as follows,

   sin x = -(0.5048 t^5)/5! - (0.8632*(t^4))/4! + (0.5048*(t^3))/3! + 0.8632*(t^2))/2! - - (0.5048*(t^1))/1! - 0.8632

3) For -1<=x<=1, sin x can be approximated as follows,

   sin x = (0.08333 * x^5)+ (0.16667 * x^3)+ x

4) For x > pi, or x < -pi, represent x as follows,

   x = n(2 * pi) + x' , where -pi <= x' <= pi and n is an integer

   Then assign x' to x and perform either one of the above steps.

The code to calculate the different trigonometric functions is given below :

```
// Function used to find the sine of a real number, and return it to any
trigonometric function
floatComp findSine(floatComp x){

  //To bring x in the range -pi < x <= pi
  while(compare(x,-1*pi)==-1)
    x=intAdd(0,x,pi);
  while(compare(x,pi)!=-1)
    x=intAdd(1,x,pi);

  floatComp temp,y,ytemp;
  floatRep(y,0);
  cout<<endl;
  floatRep(ytemp,1);
  cout<<endl;
  temp=ytemp;

  if(compare(x,-1)==-1)
  {
    /*
       For x<-1, sin x can be approximated as follows,

          sin x = -(0.5048*(t^5))/5! + (0.8632*(t^4))/4! + (0.5048*(t^3))/3!
                - (0.8632*(t^2))/2! - (0.5048*(t^1))/1! + 0.8632

    */

    x=intAdd(0,x,2.1);

    y=intAdd(1,y,0.8632035);

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,1);
    y=addab(1,y,intMultiply(0,temp,0.504802));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,2);
    y=addab(0,y,intMultiply(0,temp,0.8632035));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,3);
    y=addab(0,y,intMultiply(0,temp,0.504802));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,4);
    y=addab(1,y,intMultiply(0,temp,0.8632035));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,5);
    y=addab(1,y,intMultiply(0,temp,0.504802));
}
else if(compare(x,1)==1)
{
     /*
       For x<-1, sin x can be approximated as follows,

          sin x = -(0.5048*(t^5))/5! - (0.8632*(t^4))/4! + (0.5048*(t^3))/3!
                + (0.8632*(t^2))/2! - (0.5048*(t^1))/1! - 0.8632

    */
    x=intAdd(1,x,2.1);

    y=intAdd(0,y,0.8632035);

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,1);
    y=addab(1,y,intMultiply(0,temp,0.504802));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,2);
    y=addab(1,y,intMultiply(0,temp,0.8632035));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,3);
    y=addab(0,y,intMultiply(0,temp,0.504802));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,4);
    y=addab(0,y,intMultiply(0,temp,0.8632035));

    temp=multiplyab(0,temp,x);
    temp=intMultiply(1,temp,5);
    y=addab(1,y,intMultiply(0,temp,0.504802));
}
```

```
    else
    {
         /*
            For -1 <= x <= 1, sin x can be approximated as follows,

                sin x = (0.08333 * x^5) + (0.16667 * x^3) + x
        */
        y=addab(0,y,x);

        temp=power(x,3);
        y=addab(0,y,intMultiply(0,temp,-0.166666666666));

        temp=power(x,5);
        y=addab(0,y,intMultiply(0,temp,0.00833333333));
    }

    return y;
}

// To find sin x, given a real number x
floatComp sine(floatComp x)
{
    floatComp s=findSine(x);
    cout<<"\n\nSine = "<<floatNum(s)<<endl;

    return s;
}

// To find cos x, given a real number x
floatComp cosine(floatComp x){

    // cos x = sin (90 - x)

    x=intAdd(1,x,pi/2);
    x=intMultiply(0,x,-1);

    floatComp c=findSine(x);
    cout<<"\n\nCo-sine = "<<floatNum(c)<<endl;
    return c;
}
```

// To find tan x, given a real number x

```
floatComp tangent(floatComp x)
{
   // tan x = (sin x)/(cos x)

   floatComp t=(multiplyab(1,sine(x),cosine(x)));
   cout<<"\n\n\nTangent = "<<floatNum(t)<<endl;

   return t;
}
```

// To find cosec x, given a real number x

```
floatComp cosecant(floatComp x)
{
   // cosec x = 1/(sin x)

   floatComp t=(inverse(sine(x)));
   cout<<"\n\n\nCo-secant = "<<floatNum(t)<<endl;

   return t;
}
```

// To find sec x, given a real number x

```
floatComp secant(floatComp x)
{
   // sec x = 1/(cos x)

   floatComp t=(inverse(cosine(x)));
   cout<<"\n\n\nSecant = "<<floatNum(t)<<endl;

   return t;
}
```

// To find cot x, given a real number x

```
floatComp cotangent(floatComp x)
{
   // cot x = (cos x)/(sin x)

   floatComp t=(multiplyab(1,cosine(x),sine(x)));
   cout<<"\n\n\nCo-tangent = "<<floatNum(t)<<endl;

   return t;
}
```

Hyperbolic Functions

Hyperbolic functions include sinh x, cosh x, tanh x, cosech x, sech x, and coth x. The relationships between these hyperbolic functions correspond to those of trigonometric functions.

The function sinh u is given by,

$$\sinh u = \frac{e^u - e^{-u}}{2}$$

The function cosh u is given by,

$$\cosh u = \frac{e^u + e^{-u}}{2}$$

Both sinh x and cosh x can be found by using the exponential function e^x. Finding sinh and cosh functions, the other hyperbolic functions can be found out in a similar manner as the corresponding trigonometric functions.

tanh x = sinh x / cosh x
cosech x = 1 / sinh x
sech x = 1 / cosh x
coth x = 1 / tanh x

The code to calculate the different hyperbolic functions is given below :

```
//To calculate sinh x
floatComp sineh(floatComp x)
{
   x=exponential(x);
   cout<<"\n\n\n";
   floatComp y,temp=inverse(x);
   y=addab(1,x,temp);
   floatComp s=intMultiply(1,y,2);
   cout<<"\n\nsinh x = "<<floatNum(s)<<endl;
   return s;
}

//To calculate cosh x
floatComp cosineh(floatComp x)
{
   x=exponential(x);
```

```
    cout<<"\n\n\n";
    floatComp y,temp=inverse(x);
    y=addab(0,x,temp);
    floatComp s=intMultiply(1,y,2);
    cout<<"\n\ncosh x = "<<floatNum(s)<<endl;
    return s;
}
```

//To calculate tanh x

```
floatComp tangenth(floatComp x)
{
    floatComp t=multiplyab(1,sineh(x),cosineh(x));
    cout<<"\n\ntanh x = "<<floatNum(t)<<endl;
    return t;
}
```

//To calculate cosech x

```
floatComp cosecanth(floatComp x)
{
    floatComp cs=inverse(sineh(x));
    cout<<"cosech x = "<<floatNum(cs)<<endl;
    return cs;
}
```

//To calculate sech x

```
floatComp secanth(floatComp x)
{
    floatComp cs=inverse(cosineh(x));
    cout<<"sech x = "<<floatNum(cs)<<endl;
    return cs;
}
```

//To calculate coth x

```
floatComp cotangenth(floatComp x)
{
    floatComp cs=inverse(tangenth(x));
    cout<<"coth x = "<<floatNum(cs)<<endl;
    return cs;
}
```